Form-based Software Composition

Markus Lumpe
Iowa State University
Department of Computer Science
113 Atanasoff Hall
Ames, USA
lumpe@cs.iastate.edu

Jean-Guy Schneider
School of Infomation Technology
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
jschneider@swin.edu.au

ABSTRACT

The development of flexible and reusable abstractions for software composition has suffered from the inherent problem that reusability and extensibility are limited due to position-dependent parameters. To tackle this problem, we have defined *forms*, immutable extensible records, that allow for the specification of compositional abstractions in a language-neutral and robust way. In this paper, we present a theory of forms and show how forms can be used to represent components based on software artifacts developed for the .NET framework.

1. INTRODUCTION

In recent years, component-oriented software technology has become the major approach to facilitate the development of evolving systems [9, 17, 24, 28]. The objective of this technology is to take elements from a collection of reusable software components (i.e., *components-off-the-shelf*) and build applications by simply plugging them together.

Currently, component-based programming is mainly carried out using mainstream object-oriented languages. These languages seem to offer already some reasonable support for component-based programming (e.g. encapsulation of state and behavior, inheritance, and late binding). Unfortunately, object-oriented techniques are not powerful enough to provide flexible and typesafe component composition and evolution mechanisms, respectively. We can identify especially (a) a lack of abstractions for building and adapting class-like components in a framework or domain specific way, (b) a lack of abstractions for defining cooperation patterns, and (c) a lack of support for checking the correctness of compositions.

In order to address the key problems, various researchers have argued that it is necessary to define a language specially designed to compose software components and to base this language on a well-defined formal semantic foundation [2, 9, 12, 13, 18, 19]. But what kind of formalism should be used as a semantic foundation?

There are several plausible candidates that can serve as computational models for component-based software development. The λ-calculus, for example, has the advantage of having a well-developed theoretical foundation and being well-suited for modeling encapsulation, composition and type issues [5], but has the disadvantage of saying nothing about concurrency or communication. Process calculi such as CCS [15] and the π-calculus [16] have been developed to address just these shortcomings. Early work in the modeling of concurrent objects [20, 21] has proven CCS to be an expressive modeling tool, except that dynamic creation and communication of new communication channels cannot be directly expressed and that abstractions over the process space cannot be expressed within CCS itself, but only at a higher level. These shortcomings have been addressed by the π-calculus, which allows new names to be introduced and communicated much in the same way the λ-calculus introduces new bound names.

Unfortunately, even though both the λ-calculus and the π-calculus can be used to model composition mechanisms [27], they are inconvenient for modeling general purpose compositional abstractions due to the dependence on positional parameters. In fact, the need to use position dependent parameters results in a limited reusability and extensibility of the defined abstractions.

Dami has tackled a similar problem in the context of the λ-calculus, and has proposed λN [6, 7], a calculus in which parameters are identified by names rather than by positions. The resulting flexibility and extensibility can also be seen, for example, in XML/HTML forms, whose fields are encoded as named (rather than positional) parameters, in Python [30], where functions can be defined to take arguments by keywords, in Visual Basic [14], where *named arguments* can be used to break the order of possibly optional parameters, and in Perl [32] where it is a common technique to pass a map of name/value pairs as argument to a function or method.

Forms are immutable extensible records that define finite mappings from keys to values. They address the inherent problem that reusability and extensibility of abstractions are limited due to position-dependent parameters [11]. We argue that forms provide the means for a unifying concept

to define robust and extensible compositional abstractions. Unlike classical records, however, forms are abstract, that is, forms are used to define mappings from keys to abstract values. This is a generalization of an earlier work on forms [11, 26] and will allow us to study forms as an environment-independent framework. In order to actually use these compositional abstractions, they have to be instantiated in a concrete target system like the $\pi\mathcal{L}$-calculus [11] or the .NET framework [22]. In this paper, we will outline the basic ideas and formalisms for first-order forms, sketch some of the important issues in relation to form equivalence and normalization, and illustrate how forms can be used as a semantic foundation for component composition.

The remainder of this paper is organized as follows: in section 2, we present first-order forms. In section 3, we develop a semantics of forms. In section 4, we illustrate, how forms can be used to represent component interfaces and component interface composition of components written in the .NET-aware language C#. We conclude with a summary of the main observations and a discussion about future work in section 5.

2. FORMS

Forms are finite variable-free mappings from an infinite set of labels denoted by $\mathcal{L}$ to an infinite set of abstract values denoted by $\mathcal{V}$. The set of abstract values is a set of distinct values. We do not require any particular property except that equality and inequality are defined for all abstract values. In fact, programming values like *Strings*, *Integers*, *Names*, and even *Objects* and *Classes* are elements of $\mathcal{V}$.

The set $\mathcal{V}$ contains a distinguished element $\mathcal{E}$ – the empty value. In the context of software composition, the empty value denotes the lack of a *component service*. That is, if a given label, say l, is bound to the empty value, then the corresponding component service is either currently unavailable or not defined at all.

We use F, G, H to range over the set $\mathcal{F}$ of forms, l, m, n to range over the set $\mathcal{L}$ of labels, and a, b, c to range over the set $\mathcal{V}$ of abstract values. The set $\mathcal{F}$ of forms is defined as follows:

F	::=	$\langle\rangle$	*empty form*
	|	$F\langle l = V\rangle$	*abstract binding extension*
	|	$F \cdot F$	*polymorphic extension*
	|	$F \backslash F$	*polymorphic restriction*
	|	$F \rightarrow l$	*form dereference*
V	::=	S	*abstract scalar value*
	|	F	*nested form*
S	::=	$\mathcal{E}$	*empty value*
	|	a	*abstract value*
	|	F_l	*abstract projection*

Every form is derived from the *empty form* $\langle\rangle$, which denotes an empty component interface (i.e., a component that does not define any service). The *abstract binding extension* $F\langle l = V\rangle$ extends a given form F with exactly one binding $\langle l = V\rangle$ that either adds a fresh service, named l, or redefines an existing one. Using *polymorphic extension*, we can add or redefine a set of services. Polymorphic extension is similar to asymmetric record concatenation [4]. In fact, if the forms F and G both define a binding for the label l, then only $G's$ binding will be accessible in the polymorphic extension $F \cdot G$. The *polymorphic restriction* $F\backslash G$ denotes a form that is restricted to all bindings of F that do not occur in G with the exception that all bindings of the form $\langle l = \mathcal{E}\rangle$ and $\langle l = \langle\rangle\rangle$ in G are ignored. Finally, the *form dereference* $F \rightarrow l$ denotes the form that is bound by label l in F. Form dereference can be used to extract a form that occurs nested within an enclosing form.

In form expressions, an abstract binding extension has precedence over a polymorphic extension, a polymorphic extension has precedence over a polymorphic restriction, which in turn has precedence over form dereference. A sequence of two or more polymorphic extensions is left associative, that is, $F_1 \cdot F_2 \cdot F_3$ is equivalent to $(F_1 \cdot F_2) \cdot F_3$. The same applies to polymorphic restriction. Parenthesis may be used in form expressions in order to enhance readability or to overcome the default precedence rules.

Forms denote *component interfaces* and *component interface composition*. A component offers services using *provided ports*, and may require services using *required ports*. If a component offers a service, then this service is bound by a particular label in the form that represents the component interface. For example, if a component can provide a service A and we want to publish this service using the port name *ServiceA*, then the corresponding component interface contains a binding $\langle ServiceA = A\rangle$. In the component interface, the service A is abstract. Therefore, the client and the component (service provider) have to agree on an interpretation of service A prior to interaction. From an external observer's point of view, the interaction between the client and the component involves an opaque entity.

Required services are denoted by *abstract projections*. Given a form F, used as a deployment environment [1], and a component, which requires a service named l, we use F_l to denote the required service bound by l in the deployment environment F.

Using *binding extension* and *projection*, it is possible to construct "flat" data structures that denote both provided and requires services. However, the composition of two or more components often results in a name clash of their port names. Moreover, it is sometimes desirable to maintain the original structure of the components in the composite. To solve these kinds of problems, it is necessary to define auxiliary abstractions that encapsulate components as services. For example, a component *ComponentA* can be represented by an auxiliary service *ServiceComponentA*. Using a binding extension $\langle ServiceA = ServiceComponentA\rangle$, we can publish this service. However, this approach requires that clients have to define an additional abstraction to extract *ComponentA* encapsulated by *ServiceComponentA*.

To facilitate the specification of structured component in-

terfaces, forms can also contain *nested forms*. Like abstract values, nested forms are bound by labels. However, a projection of a nested form yields $\mathcal{E}$. The reason for this is that forms represent sets of key-value bindings and not abstract values. To extract a nested form bound by a label l in a form F, we use $F \rightarrow l$. Note that if the binding involving label l does not denote a nested form, then the actual value of $F \rightarrow l$ is $\langle\rangle$ – the empty form.

3. SEMANTICS OF FORMS

The underlying semantic model of forms is that of a record data structure. Forms are generic extensible records, where field selection is performed from right-to-left.

The interpretation of forms is defined by an evaluation function $[\![]\!]^F : \mathcal{F} \rightarrow \hat{\mathcal{F}}$, which is a total function from forms to form values. Like forms, form values are finite mappings from an infinite set of labels denoted by $\mathcal{L}$ to an infinite set of abstract values denoted by $\mathcal{V}$. However, form values do not contain any *projections* or *form dereferences*.

We use $\hat{F}, \hat{G}, \hat{H}$ to range over the set $\hat{\mathcal{F}}$ of form values, l, m, n to range over the set $\mathcal{L}$ of labels, and a, b, c to range over the set $\mathcal{V}$ of abstract values. The set $\hat{\mathcal{F}}$ of form values is a subset of $\mathcal{F}$, i.e., $\hat{\mathcal{F}} \subset \mathcal{F}$, and is defined as follows:

$\hat{F}$	::=	$\langle\rangle$	*empty form value*
	|	$\hat{F}\langle l = \hat{V}\rangle$	*binding extension value*
	|	$\hat{F} \cdot \hat{F}$	*polymorphic extension value*
	|	$\hat{F} \backslash \hat{F}$	*polymorphic restriction value*
$\hat{V}$	::=	$\hat{S}$	*abstract scalar value*
	|	$\hat{F}$	*nested form value*
$\hat{S}$	::=	$\mathcal{E}$	*empty value*
	|	a	*abstract value*

In order to define $[\![]\!]^F$, we need to define two mutually dependent functions to evaluate *projections* and *form dereferences*. The function $[\![]\!] : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \mathcal{V}$, called *projection evaluation*, is a total function from pairs $(\hat{F}, l) \in \hat{\mathcal{F}} \times \mathcal{L}$ to abstract values $a \in \mathcal{V}$, whereas the function $\langle\!\langle\rangle\!\rangle : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \hat{\mathcal{F}}$, called *form dereference evaluation*, is a total function from pairs $(\hat{F}, l) \in \hat{\mathcal{F}} \times \mathcal{L}$ to form values $\hat{G} \in \hat{\mathcal{F}}$. We define *projection evaluation* first.

DEFINITION 1. *Let $\hat{F} \in \hat{\mathcal{F}}$ be a form value and l be a label. Then the application of the function $[\![]\!] : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \mathcal{V}$ to the projection $\hat{F}_l$, written $[\![\hat{F}_l]\!]$, yields an abstract value $a \in \mathcal{V}$ and is inductively defined as follows:*

$$
\begin{array}{rcll}
[\![\langle\rangle_l]\!] & = & \mathcal{E} & \\
[\![(\hat{F}\langle m = \hat{V}\rangle)_l]\!] & = & [\![\hat{F}_l]\!] & \text{if } m \neq l \\
[\![(\hat{F}\langle l = \hat{V}\rangle)_l]\!] & = & \begin{cases} \hat{V} & \text{if } \hat{V} \in \mathcal{V} \\ \mathcal{E} & \text{otherwise} \end{cases} & \\
[\![(\hat{F} \cdot \hat{G})_l]\!] & = & \begin{cases} [\![\hat{G}_l]\!] & \text{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \\ & \quad \langle\!\langle \hat{G} \rightarrow l \rangle\!\rangle \neq \langle\rangle \\ [\![\hat{F}_l]\!] & \text{otherwise} \end{cases} & \\
[\![(\hat{F} \backslash \hat{G})_l]\!] & = & \begin{cases} \mathcal{E} & \text{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \\ & \quad \langle\!\langle \hat{G} \rightarrow l \rangle\!\rangle \neq \langle\rangle \\ [\![\hat{F}_l]\!] & \text{otherwise} \end{cases} &
\end{array}
$$

To illustrate the effect of *projection evaluation*, consider the following examples:

$$
\begin{array}{l}
[\![(\langle\rangle\langle l = a\rangle\langle m = b\rangle)_m]\!] = b \\
[\![((\langle\rangle\langle l = a\rangle\langle m = b\rangle) \backslash (\langle\rangle\langle m = c\rangle))_m]\!] = \mathcal{E} \\
[\![(\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle)_m]\!] = \mathcal{E} \\
[\![((\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \cdot (\langle\rangle\langle l = b\rangle\langle m = d\rangle))_m]\!] = d
\end{array}
$$

In the form $\langle\rangle\langle l = a\rangle\langle m = b\rangle$ the abstract value b is bound by label m. Therefore, $[\![(\langle\rangle\langle l = a\rangle\langle m = b\rangle)_m]\!]$ yields b. The second projection evaluation yields $\mathcal{E}$, because $\langle\rangle\langle m = c\rangle$ restricts $\langle\rangle\langle l = a\rangle\langle m = b\rangle$ by hiding the binding involving label m. In the third example, the label m binds a nested form. However, nested forms are not abstract values. Therefore, the result of the projection evaluation is $\mathcal{E}$. Finally, since the form $\langle\rangle\langle l = b\rangle\langle m = d\rangle$ is used as a polymorphic extension, the project evaluation of the whole form yields d, which is the right-most value bound by label m.

DEFINITION 2. *Let $\hat{F} \in \hat{\mathcal{F}}$ be a form value and l be a label. Then the application of the function $\langle\!\langle\rangle\!\rangle : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \hat{\mathcal{F}}$ to the form dereference $\hat{F} \rightarrow l$, written $\langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle$, yields a form value $\hat{G} \in \hat{\mathcal{F}}$ and is inductively defined as follows:*

$$
\begin{array}{rcll}
\langle\!\langle \langle\rangle \rightarrow l \rangle\!\rangle & = & \langle\rangle & \\
\langle\!\langle (\hat{F}\langle m = \hat{V}\rangle) \rightarrow l \rangle\!\rangle & = & \langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle & \text{if } m \neq l \\
\langle\!\langle (\hat{F}\langle l = \hat{V}\rangle) \rightarrow l \rangle\!\rangle & = & \begin{cases} \langle\rangle & \text{if } \hat{V} \in \mathcal{V} \\ \hat{V} & \text{otherwise} \end{cases} & \\
\langle\!\langle (\hat{F} \cdot \hat{G}) \rightarrow l \rangle\!\rangle & = & \begin{cases} \langle\!\langle \hat{G} \rightarrow l \rangle\!\rangle & \text{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \\ & \quad \langle\!\langle \hat{G} \rightarrow l \rangle\!\rangle \neq \langle\rangle \\ \langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle & \text{otherwise} \end{cases} & \\
\langle\!\langle (\hat{F} \backslash \hat{G}) \rightarrow l \rangle\!\rangle & = & \begin{cases} \langle\rangle & \text{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \\ & \quad \langle\!\langle \hat{G} \rightarrow l \rangle\!\rangle \neq \langle\rangle \\ \langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle & \text{otherwise} \end{cases} &
\end{array}
$$

To illustrate the effect of *dereference evaluation*, consider the following examples:

$$
\begin{array}{l}
\langle\!\langle (\langle\rangle\langle l = a\rangle\langle m = b\rangle) \rightarrow m \rangle\!\rangle = \langle\rangle \\
\langle\!\langle ((\langle\rangle\langle l = a\rangle\langle m = b\rangle) \backslash (\langle\rangle\langle m = c\rangle)) \rightarrow m \rangle\!\rangle = \langle\rangle \\
\langle\!\langle (\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \rightarrow m \rangle\!\rangle = \langle\rangle\langle n = c\rangle \\
\langle\!\langle ((\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \cdot (\langle\rangle\langle m = d\rangle)) \rightarrow m \rangle\!\rangle = \langle\rangle
\end{array}
$$

In the form $\langle\rangle\langle l=a\rangle\langle m=b\rangle$ the label m binds an abstract value. Therefore, dereference evaluation yields $\langle\rangle$. The second dereference evaluation also yields $\langle\rangle$ because $\langle\rangle\langle m=c\rangle$ restricts $\langle\rangle\langle l=a\rangle\langle m=b\rangle$ by hiding the binding involving label m. In the third example, label m binds a nested form. Hence, the result of the dereference evaluation is $\langle\rangle\langle n=c\rangle$. Finally, since the polymorphic extension $\langle\rangle\langle l=b\rangle\langle m=d\rangle$ binds the right-most label m to an abstract value, the result of the dereference evaluation of the whole form is $\langle\rangle$.

Now, we can define $[\![]\!]^F$. The application of $[\![]\!]^F$ to a given form F yields a form value $\hat{F}$.

Definition 3. *Let $F \in \mathcal{F}$ be a form. The evaluation of a form F, written $[\![F]\!]^F$, yields a form value $\hat{F} \in \hat{\mathcal{F}}$ and is inductively defined as follows:*

$$
\begin{array}{lcl@{\qquad}lcl}
[\![\langle\rangle]\!]^F & = & \langle\rangle & [\![F]\!]^V & = & [\![F]\!]^F \\
[\![F\langle l=V\rangle]\!]^F & = & [\![F]\!]^F\langle l=[\![V]\!]^V\rangle & [\![S]\!]^V & = & [\![S]\!]^S \\
[\![F\cdot G]\!]^F & = & [\![F]\!]^F\cdot[\![G]\!]^F & & & \\
[\![F\backslash G]\!]^F & = & [\![F]\!]^F\backslash[\![G]\!]^F & [\![\mathcal{E}]\!]^S & = & \mathcal{E} \\
[\![F\rightarrow l]\!]^F & = & \langle\!\langle[\![F]\!]^F\rightarrow l\rangle\!\rangle & [\![a]\!]^S & = & a \\
 & & & [\![F_l]\!]^S & = & [\![([\![F]\!]^F)_l]\!]
\end{array}
$$

The function $[\![]\!]^F$, while preserving all polymorphic extensions and polymorphic restrictions, evaluates all projections and form dereferences in F and replaces them by their corresponding value. In fact, $[\![F]\!]^F$ yields a form value $\hat{F}$ that does not contain any projection or dereference.

The effect of *form evaluation* is shown in the following example:

$$
\begin{array}{ll}
 & [\![\langle\rangle\langle l=\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k)_m\rangle\rangle]\!]^F \\
= & [\![\langle\rangle]\!]^F\langle l=[\![\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k)_m\rangle]\!]^V\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k)_m\rangle]\!]^F\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle]\!]^F\langle m=[\![(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k)_m]\!]^V\rangle\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle]\!]^F\langle m=[\![(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k)_m]\!]^S\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![([\![\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k]\!]^F)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle[\![\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle]\!]^F\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle[\![\langle\rangle]\!]^F\langle k=[\![\langle\rangle\langle m=w\rangle]\!]^V\rangle\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=[\![\langle\rangle\langle m=w\rangle]\!]^F\rangle\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=[\![\langle\rangle]\!]^F\langle m=[\![w]\!]^V\rangle\rangle\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=\langle\rangle\langle m=[\![w]\!]^S\rangle\rangle\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle\rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\rangle\langle m=w\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=w\rangle\rangle \qquad \square
\end{array}
$$

Even though a form may contain a binding for a label, say l, this form may be indistinguishable from a form that does not contain a binding for label l. In this case, we say that the label l occurs transparent, which denotes the fact that a particular service or component interface is not available.

Definition 4. *A label l is transparent with respect to form value $\hat{F}$, written $\hat{F} \parallel l$, iff*

$$[\![\hat{F}_l]\!] \;=\; \mathcal{E} \;\wedge\; [\![\hat{F}\rightarrow\; l]\!] \;=\; \langle\rangle.$$

Transparent labels can also be used for information hiding. For example, consider $\hat{F} \equiv \langle\rangle\langle l=a\rangle\langle k=(\langle\rangle\cdot\hat{G})\rangle$. We can hide the service located at label l by applying the binding extension $\langle l=\mathcal{E}\rangle$, such that $\hat{F}' \equiv \hat{F}\langle l=\mathcal{E}\rangle$ and $\hat{F}' \parallel l$. Similarly, we can hide the component interface located at label k with the binding extension $\langle k=\langle\rangle\rangle$, such that $\hat{F}'' \equiv \hat{F}\langle k=\langle\rangle\rangle$ and $\hat{F}'' \parallel k$. In both cases, the corresponding service and component interface, respectively, become inaccessible just as if the labels had never been defined.

The dual notion of $\hat{F} \parallel l$ is $\hat{F} \nparallel l$, which represents the fact that the value bound by label l in a form value $\hat{F}$ is different from $\mathcal{E}$ and $\langle\rangle$, respectively. The property $\hat{F} \nparallel l$ does not state, however, which feature is actually provided by the form value $\hat{F}$; it can be either a service or a component interface.

Definition 5. *A label l is nontransparent with respect to form value $\hat{F}$, written $\hat{F} \nparallel l$, iff*

$$[\![\hat{F}_l]\!] \;\neq\; \mathcal{E} \;\vee\; [\![\hat{F}\rightarrow\; l]\!] \;\neq\; \langle\rangle.$$

An important questions in our theory of forms is when two forms can be said to exhibit the same meaning. As in the λ-calculus, the most intuitive way of defining an equivalence of forms is via a notion of *contextual equivalence*.

A *form context* $\mathcal{C}[\cdot]$ is obtained when the hole $[\cdot]$ replaces an occurrence of a form (i.e., F) in the grammar of forms. We say that the forms F and G are equivalent, when $\mathcal{C}[F]$ and $\mathcal{C}[G]$ have the same "observable meaning" for each form context $\mathcal{C}[\cdot]$.

Definition 6. *Let $\hat{F}$ be a form value. Then the set of nontransparent labels of a form value $\hat{F}$, written $\hat{\mathcal{L}}_{\nparallel}(\hat{F})$, is defined as follows:*

$$\hat{\mathcal{L}}_{\nparallel}(\hat{F}) = \{\; l \in \mathcal{L} \mid \hat{F} \nparallel l \;\}$$

This definition gives rise to the definition of *behavioral equivalence* of forms.

Definition 7. *Two forms F and G are behaviorally equivalent, written $F \approx G$, if and only if, for all nontransparent labels $l \in \hat{\mathcal{L}}_{\nparallel}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\nparallel}([\![G]\!]^F)$,*

$$[\![([\![F]\!]^F)_l]\!] = [\![([\![G]\!]^F)_l]\!] \wedge ((([\![F]\!]^F\rightarrow\; l) \approx (([\![G]\!]^F\rightarrow\; l))$$

.

Two forms F and G are equivalent if all projection evaluations of label $l \in \hat{\mathcal{L}}_{\nparallel}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\nparallel}([\![G]\!]^F)$ yield the same value for both forms and if all their nested forms bound by label l

are equivalent. In the case of $\hat{\mathcal{L}}_{\not\Vdash}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\not\Vdash}([\![G]\!]^F) = \emptyset$, two forms F and G both evaluate to $\langle\rangle$ and they are considered equivalent.

The relation defined by $\approx$ is an equivalence relation. Furthermore, $\approx$ is preserved by all form operations, that is, $(F \approx G) \Rightarrow (\mathcal{C}[F] \approx \mathcal{C}[G])$.

Rather than defining form equivalence over all labels in $\mathcal{L}$, we restrict the equivalence of two forms F and G to the union of their nontransparent labels. This is an optimization, but it can be shown that for all labels $l \notin \hat{\mathcal{L}}_{\not\Vdash}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\not\Vdash}([\![G]\!]^F)$ both forms F and G exhibit also the same behavior.

Forms are immutable data structures. Over time, a form can grow, which results in the fact that much of its bindings become potentially inaccessible. Those bindings can be garbage collected. After garbage collection a so-called *normalized form* solely contains *binding extensions*.

We use $\bar{F}, \bar{G}, \bar{H}$ to range over the set $\bar{\mathcal{F}}$ of normalized form values. The set $\bar{\mathcal{F}}$ of normalized form values is a subset of $\hat{\mathcal{F}}$, i.e., $\bar{\mathcal{F}} \subset \hat{\mathcal{F}}$, and is defined as follows:

$$\bar{F} \quad ::= \quad \begin{cases} \langle\rangle & n = 0 \\ \langle\rangle\langle l_1 = v_1\rangle\langle l_2 = v_2\rangle \ldots \langle l_n = v_n\rangle & n > 0 \end{cases}$$

where

- all labels l_i are pairwise distinct, that is, for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, it holds that $l_i \neq l_j$, and
- each value v_i with $i \in \{1, \ldots, n\}$ is either an abstract value different from $\mathcal{E}$ or a non-empty normalized form.

With *normalized forms*, we recover classical records. However, we still maintain position independency, that is, it holds that $\langle\rangle\langle l = a\rangle\langle m = b\rangle \approx \langle\rangle\langle m = b\rangle\langle l = a\rangle$.

For every form F there exists a normalized form $\bar{F}$, such that $F \approx \bar{F}$. This normalized form can be generated by the following algorithm:

let
  Normalize($\hat{F}$, $\emptyset$) = $\langle\rangle$
  Normalize($\hat{F}$, $\{l\} \cup \hat{\mathcal{L}}$) =
    if $[\![\hat{F}_l]\!] \neq \mathcal{E}$
      then (Normalize($\hat{F}$, $\hat{\mathcal{L}}$))$\langle l = [\![\hat{F}_l]\!]\rangle$
      else
        let
          $\hat{G}$ = Normalize($\langle\!\langle \hat{F} \rightarrow\ l \rangle\!\rangle, \hat{\mathcal{L}}(\langle\!\langle \hat{F} \rightarrow\ l \rangle\!\rangle)$)
        in
          if $\hat{G} \neq \langle\rangle$
            then (Normalize($\hat{F}$, $\hat{\mathcal{L}}$))$\langle l = \hat{G}\rangle$
            else Normalize($\hat{F}$, $\hat{\mathcal{L}}$)
in
  Normalize($[\![F]\!]^F$, $\hat{\mathcal{L}}(\hat{F})$)

Using this algorithm, we can always transform a given form F into its corresponding behaviorally equivalent normalized form $\bar{F}$.

4. APPLICATION OF FORMS

Forms are used to represent both components and component interfaces. This in turn requires that forms are compile-time and run-time entities. As compile-time entities forms are used to specify component interfaces and component interface composition. At run-time, forms provide a uniform access to component services in an object-oriented way. In fact, being run-time entities forms may also allow for dynamic composition scheme or a hot-swap of components.

What is a software component? Using the characterization defined by Nierstrasz [17], a software component is a "static abstraction with plugs". But this characterization is rather vague. In this paper, we propose a new paradigm that is based on *module interconnection languages* [8] and *traits* [25].

In our new paradigm, a component is a collection of cooperating objects, each representing a partial state of the component. This approach stresses the view that, in general, we need a set of programming entities (or objects) to represent one component. We consider a one-to-one relationship between an object and a component (i.e., one object completely implements the semantics of a component) a special case. This is somewhat a departure from the approach mostly used today to represent components on top of an object-oriented programming language or system.

The provided and required services of a component are modeled by method pointers (or delegates) and forms are used to represent component interfaces. It is also possible to specify multiple forms (i.e., component interfaces) for the same collection of cooperating objects (i.e., the component), by allowing that different interfaces can share the same delegates. Using this approach, we can think of forms as traits with the exception that the services specified in forms have also an associated state. In fact, forms provide an abstraction similar to modules and the form operations can be used to combine modules.

In order to illustrate our approach, we will use some software artifacts developed in the .NET framework. In particular, we will show how these artifacts (i.e., C# code) can be encapsulated by forms using delegates as instantiations of abstract values. We can do so, because in the .NET framework both (structural) equality and inequality are defined for delegates.

Delegates are one of the most notable innovations of the C# language and the .NET framework. Delegates are type-safe, secure managed objects that behave like *method pointers* [3, 10, 29]. A delegate is a reference type that can encapsulate a method with a particular signature and a return type. In the .NET framework, delegates are mainly used to specify events and callbacks.

Delegates are defined using the `delegate` keyword, followed by a return type and a method signature. For example, consider the following delegate declarations:

```
public delegate void Setter( Object aValue );
public delegate Object Getter();
```

These declarations define the delegates `Setter` and `Getter`, which can encapsulate any method that takes an object as parameter and has the return type `void` (`Setter`), or has no parameters and returns an object (`Getter`).

However, while primarily being used for events and callbacks, delegates also provide a level of abstraction that enables us to use them to represent component plugs. In fact, a plug can be considered as callback that, when notified, provides a service or requires a service in turn.

Now, consider a generic *storage cell*, which represents an updatable data structure. A storage cell maintains a private state (i.e., its contents), and has at least two methods: `get` to read its contents and `set` to update its contents, respectively. A generic[1] C# implementation is shown in the following:

```
// generic reference cell
class StorageCell
{
  // Contents
  private Object fContents;

  // Getter method
  public Object get()
    { return fContents; }

  // Setter method
  public void set( Object aValue )
    { fContents = aValue; }
}
```

Now, this C# storage cell can be represented by the following form:

$$StorageCell \equiv \langle\rangle\langle get = \text{aObjGetter}\rangle\langle set = \text{aObjSetter}\rangle$$

In the form *StorageCell*, both `aObjGetter` and `aObjSetter` are delegates. The required C# code to define both delegates is shown in the following code fragment:

```
// create a fresh storage cell object
StorageCell aObj = new StorageCell();

// create the delegates
Setter aObjSetter = new Setter( aObj.set );
Getter aObjGetter = new Getter( aObj.get );
```

To set the contents of our storage cell component to the string value ``A new string value'', we can use the following pseudo-code expression:

$$[\![([\![StorageCell]\!]^F)_{set}]\!](\text{ “A new string value” })$$

In this expression, $[\![([\![StorageCell]\!]^F)_{set}]\!]$ yields the delegate `aObjSetter`, which, when applied to the string argument, invokes `aObj`'s `set` method.

Similarly, to extract the current contents from our storage cell component, we can use

$$[\![([\![StorageCell]\!]^F)_{get}]\!]()$$

in which, $[\![([\![StorageCell]\!]^F)_{get}]\!]$ yields the `aObjGetter` delegate that, when called, invokes `aObj`'s `get` method.

In a second example, we illustrate the composition of two components using an approach similar to COM/ActiveX aggregation [23]. Suppose we have a `Multiselector` and a `GUIList` component. The `GUIList` component provides two services `paint` and `close` whereas the `Multiselector` provides the services `select`, `deselect`, and `close`. Both components can be represented by the following forms:

$$Multiselector \equiv \langle\rangle\langle select = \text{s}\rangle\langle deselect = \text{d}\rangle\langle close = \text{c}\rangle$$
$$GUIList \equiv \langle\rangle\langle paint = \text{p}\rangle\langle close = \text{c2}\rangle$$

where `s`, `d`, `c`, `p`, and `c2` are delegates. The required C# code to define the delegates is shown in the following code fragment:

```
// delegate type definition
public delegate void GuiOp();

// GUI objects
Multiselector aMultiselector =
                  new Multiselector();
GUIList aGUIList = new GUIList();

// plug delegates
GuiOp s = new GuiOp( aMultiselector.select );
GuiOp d = new GuiOp( aMultiselector.deselect );
GuiOp c = new GuiOp( aMultiselector.close );
GuiOp p = new GuiOp( aGUIList.paint );
GuiOp c2 = new GuiOp( aGUIList.close );
```

A composition of these two components has to offer the union of both sets of services, and, in order to close the composite component correctly, an invocation of `close` must be forwarded to both components. In order to define the required dispatch of `close`, we have to define some *glue code*. That is, we construct a new delegate `dispatchClose` and register the delegates yielded by the projection evaluations $[\![([\![Multiselector]\!]^F)_{close}]\!]$ and $[\![([\![GUIList]\!]^F)_{close}]\!]$:

$$\texttt{dispatchClose} \equiv [\![([\![Multiselector]\!]^F)_{close}]\!] + [\![([\![GUIList]\!]^F)_{close}]\!];$$

In this glue pseudo-code, the delegate `dispatchClose` is actually a multicast delegate that, when called, invokes all registered delegates.

[1]The .NET framework has a fully object-oriented data model, where every data type is derived from `Object`. Therefore, we can use this data type to enable a storage cell to hold values of any .NET data type.

Using the newly defined delegate, we can define our first composite `fixedcompose`:

$$
\begin{aligned}
&fixedcompose \equiv \\
&\quad \langle\rangle \\
&\quad \langle select = [\![([\![Multiselector]\!]^F)_{select}]\!]\rangle \\
&\quad \langle deselect = [\![([\![Multiselector]\!]^F)_{deselect}]\!]\rangle \\
&\quad \langle paint = [\![([\![GUIList]\!]^F)_{paint}]\!]\rangle \\
&\quad \langle close = \text{dispatchClose}\rangle
\end{aligned}
$$

However, even though `fixedcompose` represents a composite component with the required functionality, it is not flexible enough. In fact, it can only work on instances like `Multiselector` and `GUIList`. If applied to components that provide additional services, then `fixcompose` will simply discard them.

To address this problem, we can define a new form, called `flexcompose`, that provides a generic abstraction, which, unlike `fixedcompose`, can work on instances that provide additional like `resize` or `selectall`.

$$
\begin{aligned}
&flexcompose \equiv \\
&\quad (Multiselector \cdot GUIList)\langle close = \text{dispatchClose}\rangle
\end{aligned}
$$

The polymorphic extension $(Multiselector \cdot GUIList)$ defines an aggregation of the components `Mulitselector` and `GUIList`. Moreover, the use of polymorphic extension guarantees that even if `Multiselector` and `GUIList` provide additional services, `flexcompose` will not discard them.

The proper dispatch of `close` is guaranteed by the binding extension $\langle close = \text{dispatchClose}\rangle$. Since it is the right-most binding extension, it hides all bindings involving label `close` in $(Multiselector \cdot GUIList)$. Therefore, an application of the `close` service $[\![([\![flexcompose]\!]^F)_{close}]\!]()$ will have the desired effect.

5. CONCLUSION AND FUTURE WORK

We have presented a small theory of forms, a special notion of immutable extensible records. Forms are the key concepts for extensibility, flexibility, and robustness in component-based application development. Furthermore, forms enable the definition of a canonical set compositional abstractions in an uniform framework.

In this paper, we have focused on the representation of components and component interfaces based on a notion of symbols. In fact, there exists a close relationship between forms and XML [31]. Both forms and XML can be used as platform-independent specifications of data types (e.g. component interfaces). In order to provide more expressiveness of forms, we are currently working of a notion of binding extension that also incorporate attribute specification that will enable us to specify additional functional and non-functional properties of services.

Forms are very similar to *traits* [25]. However, unlike traits forms incorporate the notion of state, if services are represented by delegates. On the other hand, both traits and forms do not affect the semantics of the underlying program entities and their composition mechanisms have similar effects.

Furthermore, we have shown that forms can be used to represent components written in C#. In fact, we have presented a new paradigm that characterizes components as collections of cooperating objects. The main idea of this approach is the use of delegates as component plugs. In fact, together with the notion of forms, delegates are most useful to define robust and reusable software abstractions.

However, forms do not exist in isolation. In fact, forms have to be embedded into a concrete computational model like the λ-calculus or the π-calculus. Then it will be possible to define true, parameterized compositional abstractions.

Future work will also include the definition of an appropriate typing scheme for forms, as types impose constraints which help to enforce the correctness of a program [5]. The plugs of a component constitute essentially an interface, or a *contractual specification*. Ideally, all conditions of a contract should be stated explicitly and formally as part of an interface specification. Furthermore, it would be highly desirable to have tools to check automatically clients and providers against the contractual specifications and in the case of a violation to reject the interaction of both.

The problem of inferring a contractual type for a given component A can be stated as the problem of finding (i) a type P that represents what component A provides, (ii) a type R that represents what component A requires of a deployment environment, and (iii) a set of constraints C, which must be satisfied by provided type P due to requirements posed by R. That is, whenever it is possible to infer (or prove the existence of) P, R, and C, software components can be safely composed.

6. ACKNOWLEDGMENTS

We would like to thank Gary Leavens for inspiring discussions on these topics as well as the anonymous reviewers for commenting on an earlier draft.

7. REFERENCES

[1] F. Achermann and O. Nierstrasz. Explicit Namespaces. In J. Gutknecht and W. Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–99. Springer, Sept. 2000.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[3] T. Archer. *Inside C#*. Mircosoft Press, 2001.

[4] L. Cardelli and J. C. Mitchell. Operations on Records. In C. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[6] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[7] L. Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, Feb. 1998.

[8] F. DeRemer and H. H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, SE–2(2):80–86, June 1976.

[9] G. Leavens and M. Sitamaran, editors. *Foundations of Component-Based Systems*. Cambridge University Press, Mar. 2000.

[10] J. Liberty. *Programming C#*. O'Reilly, 2nd edition, 2002.

[11] M. Lumpe. *A π-Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 1999.

[12] M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a formal composition language. In G. T. Leavens and M. Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, Sept. 1997.

[13] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proceedings ESEC '95*, LNCS 989, pages 137–153. Springer, Sept. 1995.

[14] Microsoft Corporation. *Visual Basic Programmierhandbuch*, 1997.

[15] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[17] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[18] O. Nierstrasz and T. D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[19] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[20] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79. Springer, 1992.

[21] M. Papathomas. Behaviour Compatibility and Specification for Active Objects. In D. Tsichritzis, editor, *Object Frameworks*, pages 31–40. Centre Universitaire d'Informatique, University of Geneva, July 1992.

[22] J. Richter. *Applied Microsoft .NET Framework Programming*. Mircosoft Press, 2002.

[23] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[24] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[25] N. Schärli, D. Stéphane, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In L. Cardelli, editor, *Proceedings of the ECOOP '03*, LNCS 2743, pages 248–274. Springer, July 2003.

[26] J.-G. Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Oct. 1999.

[27] J.-G. Schneider and M. Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In R. Ducournau and S. Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, Oct. 1997. Hermes.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[29] A. Troelsen. *C# and the .NET Platform*. Apress, 2001.

[30] G. van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), Oct. 1996.

[31] W3C Recommendation. *Extensible Markup Language (XML) 1.0 (Second Edition)*, Oct. 2000. http://www.w3.org/TR/REC-xml.

[32] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, Sept. 1996.